```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import random

import matplotlib.pyplot as plt

# Features: inlet_temp, feed_flow_rate, atomizer_speed
# Target: moisture_content

np.random.seed(42)
X = np.random.uniform(low=150, high=190, size=(100, 3))  # Simulated
inputs
y = 0.03 * X[:, 0] - 0.02 * X[:, 1] + 0.01 * X[:, 2] +
np.random.normal(0, 1, 100)  # Simulated moisture content

scaler_X = MinMaxScaler()
scaler_y = MinMaxScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).flatten()

def build_ann(weights):
    model = Sequential()
    model.add(Dense(6, input_dim=3, activation='relu'))
    model.add(Dense(1, activation='linear'))

    # Set weights from GA
    idx = 0
    for layer in model.layers:
        shapes = [w.shape for w in layer.get_weights()]
        weights_layer = []
        for shape in shapes:
            size = np.prod(shape)
            weights_layer.append(np.array(weights[idx:idx +
size]).reshape(shape))
            idx += size
        layer.set_weights(weights_layer)

    return model

# GA parameters
POP_SIZE = 30
NUM_GEN = 50
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.05

# Get weight dimensions
```

```python
model_template = Sequential([
    Dense(6, input_dim=3, activation='relu'),
    Dense(1, activation='linear')
])
num_weights = sum([np.prod(w.shape) for layer in model_template.layers
for w in layer.get_weights()])

# Initialize population
def init_population():
    return [np.random.uniform(-1, 1, num_weights) for _ in
range(POP_SIZE)]

# Fitness function
def fitness(individual):
    model = build_ann(individual)
    model.compile(loss='mse', optimizer='adam')
    y_pred = model.predict(X_scaled, verbose=0).flatten()
    return -mean_squared_error(y_scaled, y_pred)  # Negative because
GA maximizes fitness

def selection(pop, scores):
    idx = np.argsort(scores)[-2:]  # Best two
    return [pop[idx[0]], pop[idx[1]]]

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, len(parent1) - 2)
        child1 = np.concatenate([parent1[:point], parent2[point:]])
        child2 = np.concatenate([parent2[:point], parent1[point:]])
        return child1, child2
    return parent1.copy(), parent2.copy()

def mutate(individual):
    for i in range(len(individual)):
        if random.random() < MUTATION_RATE:
            individual[i] += np.random.normal(0, 0.1)
    return individual

population = init_population()
for gen in range(NUM_GEN):
    scores = [fitness(ind) for ind in population]
    best_score = max(scores)
    print(f"Generation {gen+1}: Best Fitness = {best_score:.4f}")

    # Select best parents
    parents = selection(population, scores)

    # Generate new population
    new_pop = []
    while len(new_pop) < POP_SIZE:
```

```
        p1, p2 = random.choices(parents, k=2)
        c1, c2 = crossover(p1, p2)
        new_pop.extend([mutate(c1), mutate(c2)])
    population = new_pop[:POP_SIZE]
```

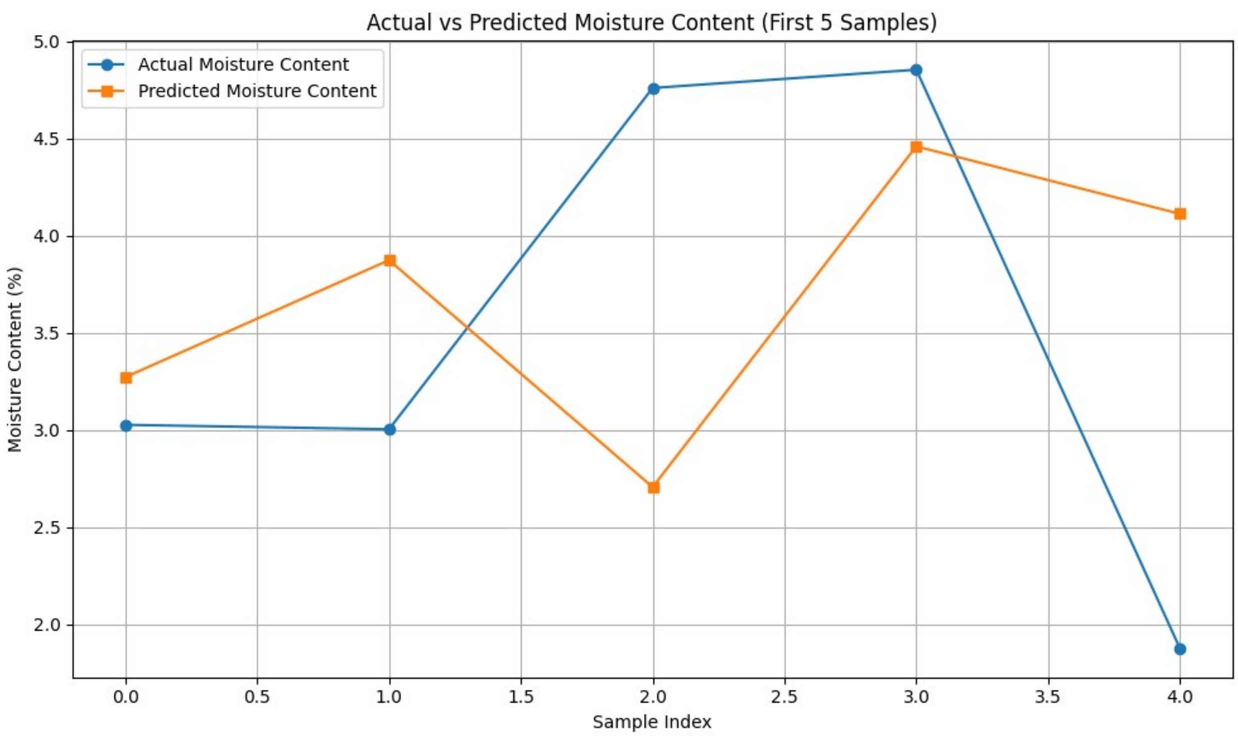
```
Generation 1: Best Fitness = -0.0508
Generation 2: Best Fitness = -0.0544
Generation 3: Best Fitness = -0.0477
Generation 4: Best Fitness = -0.0445
Generation 5: Best Fitness = -0.0405
Generation 6: Best Fitness = -0.0318
Generation 7: Best Fitness = -0.0298
Generation 8: Best Fitness = -0.0295
Generation 9: Best Fitness = -0.0289
Generation 10: Best Fitness = -0.0289
Generation 11: Best Fitness = -0.0276
Generation 12: Best Fitness = -0.0272
Generation 13: Best Fitness = -0.0268
Generation 14: Best Fitness = -0.0264
Generation 15: Best Fitness = -0.0261
Generation 16: Best Fitness = -0.0260
Generation 17: Best Fitness = -0.0257
Generation 18: Best Fitness = -0.0256
Generation 19: Best Fitness = -0.0255
Generation 20: Best Fitness = -0.0252
Generation 21: Best Fitness = -0.0252
Generation 22: Best Fitness = -0.0246
Generation 23: Best Fitness = -0.0242
Generation 24: Best Fitness = -0.0242
Generation 25: Best Fitness = -0.0242
Generation 26: Best Fitness = -0.0242
Generation 27: Best Fitness = -0.0241
Generation 28: Best Fitness = -0.0240
Generation 29: Best Fitness = -0.0240
Generation 30: Best Fitness = -0.0240
Generation 31: Best Fitness = -0.0236
Generation 32: Best Fitness = -0.0235
Generation 33: Best Fitness = -0.0234
Generation 34: Best Fitness = -0.0232
Generation 35: Best Fitness = -0.0231
Generation 36: Best Fitness = -0.0230
Generation 37: Best Fitness = -0.0229
Generation 38: Best Fitness = -0.0229
Generation 39: Best Fitness = -0.0228
Generation 40: Best Fitness = -0.0228
Generation 41: Best Fitness = -0.0228
Generation 42: Best Fitness = -0.0227
Generation 43: Best Fitness = -0.0225
Generation 44: Best Fitness = -0.0225
Generation 45: Best Fitness = -0.0225
```

```
Generation 46: Best Fitness = -0.0224
Generation 47: Best Fitness = -0.0224
Generation 48: Best Fitness = -0.0224
Generation 49: Best Fitness = -0.0223
Generation 50: Best Fitness = -0.0223
```

```
final_scores = [fitness(ind) for ind in population]
best_idx = np.argmax(final_scores)
best_weights = population[best_idx]

best_model = build_ann(best_weights)
best_model.compile(loss='mse', optimizer='adam')
y_pred = best_model.predict(X_scaled, verbose=0)
y_pred_actual = scaler_y.inverse_transform(y_pred)

print("\nPredicted Moisture Content (sample):")
print(y_pred_actual[:5])
```

```
Predicted Moisture Content (sample):
[[3.2726817]
 [3.8747573]
 [2.7066998]
 [4.4604006]
 [4.113116 ]]
```

```
# If you have actual moisture values (before scaling), use them.
Otherwise, use the first 5 for illustration
y_actual = scaler_y.inverse_transform(y_scaled.reshape(-1, 1))  #
Actual moisture content

# Select first 5 samples for visualization
actual_values = y_actual[:5].flatten()
predicted_values = y_pred_actual[:5].flatten()

# Plot actual vs predicted moisture content
plt.figure(figsize=(10, 6))
plt.plot(actual_values, label='Actual Moisture Content', marker='o')
plt.plot(predicted_values, label='Predicted Moisture Content',
marker='s')
plt.title('Actual vs Predicted Moisture Content (First 5 Samples)')
plt.xlabel('Sample Index')
plt.ylabel('Moisture Content (%)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```


Actual vs Predicted Moisture Content (First 5 Samples)
Actual Moisture Content
Predicted Moisture Content
Moisture Content (%)
Sample Index
5.0
4.5
4.0
3.5
3.0
2.5
2.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0